String Matching Problem and Variants

Aman Goel, Undergraduate, Indian Institute of Technology Madras

(Review Paper as a part of course EE4371: Introduction to DSA)

***Abstract*— String matching problem is the problem of finding all occurrences of a given pattern string in a text. In this report, we review the Knuth-Morris-Pratt (KMP) algorithm that solves the string matching problem in linear time. We will first look into the history of string matching algorithms, KMP algorithm, brief explanation and running time analysis. We then look briefly into other similar algorithms and real world applications of string matching algorithms.**

I. Introduction

Frequently in text editing programs, it is required to search for all occurrences of a pattern string (inputted by user) in a text document. Use of fast and efficient algorithms can highly improve the response time of text editing programs. Knuth-Morris-Pratt algorithm is the first string matching algorithm that achieves linear running time. The algorithm was developed by D. Knuth, V. Pratt and J. H. Morris in 1977. Many other variants have been developed to improve the efficiency in different applications.

II. Problem Definition

The string matching problem can be formulated as follows:

Pattern to be searched is an array P[m] of length m and text (document) is an array T[n] of length n. Elements of P and T are characters belonging to finite set Σ (example Σ = {a, b,.., z}). The problem is to find all $s \in [0, n-m]$ such that $T[s + i] = P[i]$ for all $i \in [1, m]$.

III. History

We first discuss the naïve brute force algorithm with complexity of O(mn). The algorithm runs a loop that matches the pattern with text for every possible value of s (valid shift). The algorithm involves an implicit loop that checks on each character position until all matches successfully or a match is found, thus making the overall complexity to O(mn).

Then came the Rabin-Karp algorithm which uses the concept similar to hashing that brings the average linear running time if expected number of occurrences is small (O(1)).

Rabin-Karp algorithm was followed by string matching with finite automata. The algorithm first preprocesses the input pattern to finite automaton by computing the transition function δ in $O(m|\Sigma|)$. It then requires one scan of the text, bringing the total running time to $O(n + m|\Sigma|)$.

IV. Knuth-Morris-Pratt algorithm

KMP algorithm keeps the information wasted by naïve approach during the text scanning and uses information learnt by previous symbol comparisons. The algorithm is similar and simpler than string matching with finite automata. This algorithm does pattern matching by preprocessing an auxiliary function $\pi[m]$, thus avoiding computation of the transition function δ altogether.

Algorithm 1 Knuth-Morris-Pratt Algorithm
Require: P, T

```
KMP-Matcher (T, P)
1    n ← length [T]
2    m ← length [P]
3    π ← Compute-Prefix (P)
4    j ← 0
5    for i from 1 to n
6        do while j > 0 and P [j + 1] ≠ T[i]
7            do j ← π[j]
8          if P [j + 1] = = T[i]
9            then j ← j + 1
10       if j = = m
11           then print "String found at", (i – m)
12                j ← π[j]

Compute-Prefix (P)
1    m ← length [P]
2    π [1] ← 0
3    k ← 0
4    for j from 2 to m
5        do while k > 0 and P [k + 1] ≠ P[j]
6            do k ← π[k]
7          if P [k + 1] = = P[j]
8            then k ← k + 1
9          π [j] ← k
10   return π
```

Given a pattern P, the algorithm first computes $\pi[j]$, the longest prefix of P that is proper suffix of P_j using the prefix function. The program then runs a simple loop that parses T and looks for its matching with P[j + 1] while keeping j updated using $\pi[j]$.

A. Correctness of prefix function computation

In Compute-Prefix function, $\pi[1] = 0$ in line 2 is correct, as $j < \pi[j]$ for every j. Lines 2-3, 9 ensures $k = \pi[j - 1]$. Lines 5-6 modifies k such that it becomes the correct value of $\pi[j]$ by searching through all values of k until one is found for which

P[k + 1] = P[j]. If there is no such k, then lines 7-9 sets π[j] to 0.

B. Brief explanation

Initially, j remains zero until a matched symbol with P[1] is found using line 8. Line 9 keeps incrementing j until a mismatch is found. If no mismatch is found, line 10-11 prints the position as answer (it is trivial to see that this is a solution). If a mismatch occurs, control enters the loop at lines 6-7 which puts π[j]'s value into j. This ensures that π[j] number of unnecessary comparisons are avoided and next comparison happens with P[π[j] + 1]. The program repeats the process till i reach n, at which point the control reaches end of T.

C. Running time analysis

In Compute-Prefix-Function, we know π[j] < j (can be shown by amortized analysis) and thus reduces lines 5-9 to O(1). Thus its running time is O(m). Similar analysis for KMP-Matcher shows that lines 6-12 has complexity of O(1). Thus its running time is O(n) summing up total running time of the algorithm to O(m + n).

V. Development and Variants

A. Long patterns and Σ

For long patterns and Σ, Boyer-Moore algorithm gives much better efficiency compared to other string matching algorithms. The program involves two heuristics that allows the program to skip many text characters altogether. The algorithm makes successive comparisons from right to left. When a mismatch occurs, both heuristics proposes a value (maximum of which is chosen) by which shift is increased without skipping any valid shift.

B. Repetition Factors

An efficient algorithm for string matching based on repetition factors was developed by Galil and Seiferas. The algorithm has linear running time complexity and requires only O(1) storage beyond P and T.

C. Approximate String Matching

The Bitap algorithm performs approximate string matching based on Levenshtein distance between strings. The algorithm requires much lesser preprocessing and can uses mostly bitwise operations, making the algorithm extremely fast.

D. Dictionary Matching

Aho-Corasick algorithm can perform multiple (but finite) pattern matching in a text in parallel achieving linear running time.

E. Polymorphic String Matching

Combination of more than one string matching algorithm (example KMP and Boyer-Moore fusion) can be used to provide a better functional algorithm with decreased space and time complexity.

VI. Applications

A. Text Editor and Search Engines

Text editors, search engines and digital libraries need to perform pattern matching in a text or database (example Turnitin). Most text editors use direct implementation of Boyer-Moore algorithm to implement find/replace command.

B. Computational Biology and Bioinformatics

String matching algorithms are widely used in DNA sequencing, finding close mutation, searching antimicrobial structures, developing local data warehouses for DNA, genes and proteins.

C. Network Intrusion Detection System

Modern intrusion and computer virus detection systems incorporate use of string matching algorithms of packets against signatures. Multiple patterns from a virus database can be matched in parallel and compiled automaton stored for later use.

D. Unix Command fgrep

Fgrep introduced in Version 7 UNIX uses Aho-Corasick algorithm to search a set of fixed strings.

E. Musical Pattern Detection

Approximate string matching algorithms are used in modern music search technique to retrieve musical note from musical database.

F. Combinatorial Universal Denoising

Algorithms like DUDE used in combinatorial denoising uses string matching algorithms to come up with distribution of occurrences of its context.

VII. Conclusion

String matching algorithms have diverse applications and have improved many complex pattern detection systems in recent years. The algorithm employed may differ based on the problem but the concept still remains similar. String matching algorithms plays a vital role in finding the appropriate match in minimum time.

References

[1] Thomas. H. Cormen, Charles. E. Leiserson, Ronald. L. Rivest, "String Matching," in *Introduction to Algorithms. Cambridge, Massachusetts: MIT Press. pp 853-885*

[2] Donald Knuth, James H. Morris, Vaughan Pratt (1977). "Fast pattern matching in strings". *SIAM Journal on Computing 6(2):323-250*

[3] Robert S. Boyer, Strother J. Moore. (1977, October). "A fast string searching algorithm". *Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20(10): 762-772*

[4] Coit, C.J., Staniford S., McAlerney, J. (2001). "Towards faster string matching for intrusion detection or exceeding the speed of Snort". Presented at DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01. Proceedings (Volume:1). IEEE (Anaheim, CA).

[5] S. Chen, S. N. Diggavi, S. Dusad, Muthukrishnan S. (2005, March). "Efficient String Matching Algorithms for Combinatorial Universal Denoising". *IEEE Data Compression Conference (DCC) (Snow Bird, UTAH).*